École Normale Supérieure

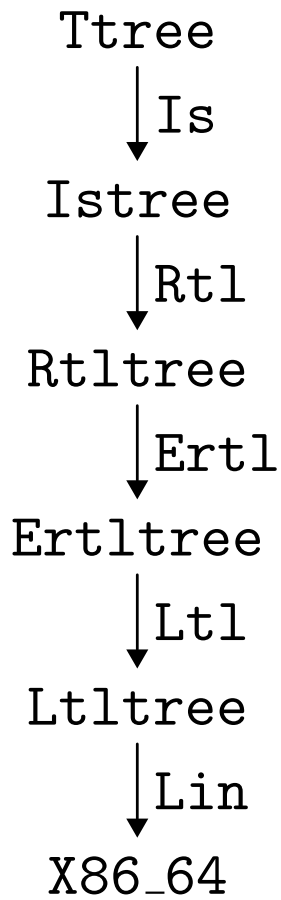
Langages de programmation et compilation

Jean-Christophe Filliâtre

compilateur optimisant (2/2)

la production de code optimisé a été découpée en plusieurs phases :

1. sélection d'instructions
2. RTL (*Register Transfer Language*)
3. ERTL (*Explicit Register Transfer Language*)
4. LTL (*Location Transfer Language*)
   - 4.1 analyse de durée de vie
   - 4.2 construction d'un graphe d'interférence
   - 4.3 allocation de registres par coloration de graphe
5. code linéarisé (assembleur)



on a pris en exemple un fragment simple de C

```
int fact(int x) {
  if (x <= 1) return 1;
  return x * fact(x-1);
}
```

phase 1 : la sélection d'instructions

```
int fact(int x) {
  if (Mjlei 1 x) return 1;
  return Mmul x fact((Maddi -1) x);
}
```

phase 2 : RTL (*Register Transfer Language*)

```
#2 fact(#1)
  entry : L10
  exit  : L1
  locals:
  L10: mov #1 #6  --> L9
  L9 : jle $1 #6  --> L8, L7
  L8 : mov $1 #2  --> L1
```

```
  L7: mov #1 #5             --> L6
  L6: add $-1 #5            --> L5
  L5: #3 <- call fact(#5) --> L4
  L4: mov #1 #4             --> L3
  L3: mov #3 #2             --> L2
  L2: imul #4 #2            --> L1
```

phase 3 : ERTL (*Explicit Register Transfer Language*)

```
fact(1)
  entry : L17
  locals: #7,#8
  L17: alloc_frame  --> L16
  L16: mov %rbx #7  --> L15
  L15: mov %r12 #8  --> L14
  L14: mov %rdi #1  --> L10
  L10: mov #1 #6    --> L9
  L9 : jle $1 #6 --> L8, L7
  L8 : mov $1 #2    --> L1
  L1 : goto         --> L22
  L22: mov #2 %rax  --> L21
  L21: mov #7 %rbx  --> L20
```

```
  L20: mov #8 %r12  --> L19
  L19: delete_frame --> L18
  L18: return
  L7 : mov #1 #5    --> L6
  L6 : add $-1 #5   --> L5
  L5 : goto         --> L13
  L13: mov #5 %rdi  --> L12
  L12: call fact(1) --> L11
  L11: mov %rax #3  --> L4
  L4 : mov #1 #4    --> L3
  L3 : mov #3 #2    --> L2
  L2 : imul #4 #2   --> L1
```

phase 4 : LTL (*Location Transfer Language*)

on a déjà réalisé l'**analyse de durée de vie** *i.e.* on a déterminé pour chaque variable (pseudo-registre ou registre physique) à quels moments la valeur qu'elle contient peut être utilisée dans la suite de l'exécution

analyse de durée de vie

```
L17: alloc_frame --> L16  in = %r12,%rbx,%rdi  out = %r12,%rbx,%rdi
L16: mov %rbx #7 --> L15  in = %r12,%rbx,%rdi  out = #7,%r12,%rdi
L15: mov %r12 #8 --> L14  in = #7,%r12,%rdi    out = #7,#8,%rdi
L14: mov %rdi #1 --> L10  in = #7,#8,%rdi      out = #1,#7,#8
L10: mov #1 #6   --> L9   in = #1,#7,#8        out = #1,#6,#7,#8
L9 : jle $1 #6 -> L8, L7  in = #1,#6,#7,#8     out = #1,#7,#8
L8 : mov $1 #2   --> L1   in = #7,#8           out = #2,#7,#8
L1 : goto        --> L22  in = #2,#7,#8        out = #2,#7,#8
L22: mov #2 %rax --> L21  in = #2,#7,#8        out = #7,#8,%rax
L21: mov #7 %rbx --> L20  in = #7,#8,%rax      out = #8,%rax,%rbx
L20: mov #8 %r12 --> L19  in = #8,%rax,%rbx    out = %r12,%rax,%rbx
L19: delete_frame--> L18  in = %r12,%rax,%rbx  out = %r12,%rax,%rbx
L18: return               in = %r12,%rax,%rbx  out =
L7 : mov #1 #5   --> L6   in = #1,#7,#8        out = #1,#5,#7,#8
L6 : add $-1 #5  --> L5   in = #1,#5,#7,#8     out = #1,#5,#7,#8
L5 : goto        --> L13  in = #1,#5,#7,#8     out = #1,#5,#7,#8
L13: mov #5 %rdi --> L12  in = #1,#5,#7,#8     out = #1,#7,#8,%rdi
L12: call fact(1)--> L11  in = #1,#7,#8,%rdi   out = #1,#7,#8,%rax
L11: mov %rax #3 --> L4   in = #1,#7,#8,%rax   out = #1,#3,#7,#8
L4 : mov #1 #4   --> L3   in = #1,#3,#7,#8     out = #3,#4,#7,#8
L3 : mov #3 #2   --> L2   in = #3,#4,#7,#8     out = #2,#4,#7,#8
L2 : imul #4 #2  --> L1   in = #2,#4,#7,#8     out = #2,#7,#8
```

on va maintenant construire un **graphe d'interférence** qui exprime les contraintes sur les emplacements possibles pour les pseudo-registres

Définition (interférence)

On dit que deux variables v_1 et v_2 **interfèrent** *si elles ne peuvent pas être réalisées par le même emplacement (registre physique ou emplacement mémoire).*

comme l'interférence n'est pas décidable, on va se contenter de conditions suffisantes

soit une instruction qui **définit** une variable v : toute autre variable w vivante à la **sortie** de cette instruction peut interférer avec v

cependant, dans le cas particulier d'un instruction

`mov` w v

on ne souhaite pas déclarer que v et w interfèrent car il peut être précisément intéressant de réaliser v et w par le même emplacement et d'éliminer ainsi une ou plusieurs instructions

on adopte donc la définition suivante

Définition (graphe d'interférence)

Le **graphe d'interférence** *d'une fonction est un graphe non orienté dont les sommets sont les variables de cette fonction et dont les arêtes sont de deux types : interférence ou préférence.*
Pour chaque instruction qui définit une variable v et dont les variables vivantes en sortie, autres que v, sont $w_1, \ldots, w_n$, on procède ainsi :

- *si l'instruction n'est pas une instruction* `mov` w v*, on ajoute les n arêtes d'interférence $v - w_i$*
- *s'il s'agit d'une instruction* `mov` w v*, on ajoute les arêtes d'interférence $v - w_i$ pour tous les w_i différents de w et on ajoute l'arête de préférence $v - w$.*

(si une arête $v - w$ est à la fois de préférence et d'interférence, on conserve uniquement l'arête d'interférence)

le graphe d'interférence peut être ainsi représenté en OCaml :

```
type arcs = { prefs: Register.set; intfs: Register.set }

type graph = arcs Register.map
```

la fonction qui le construit est de la forme

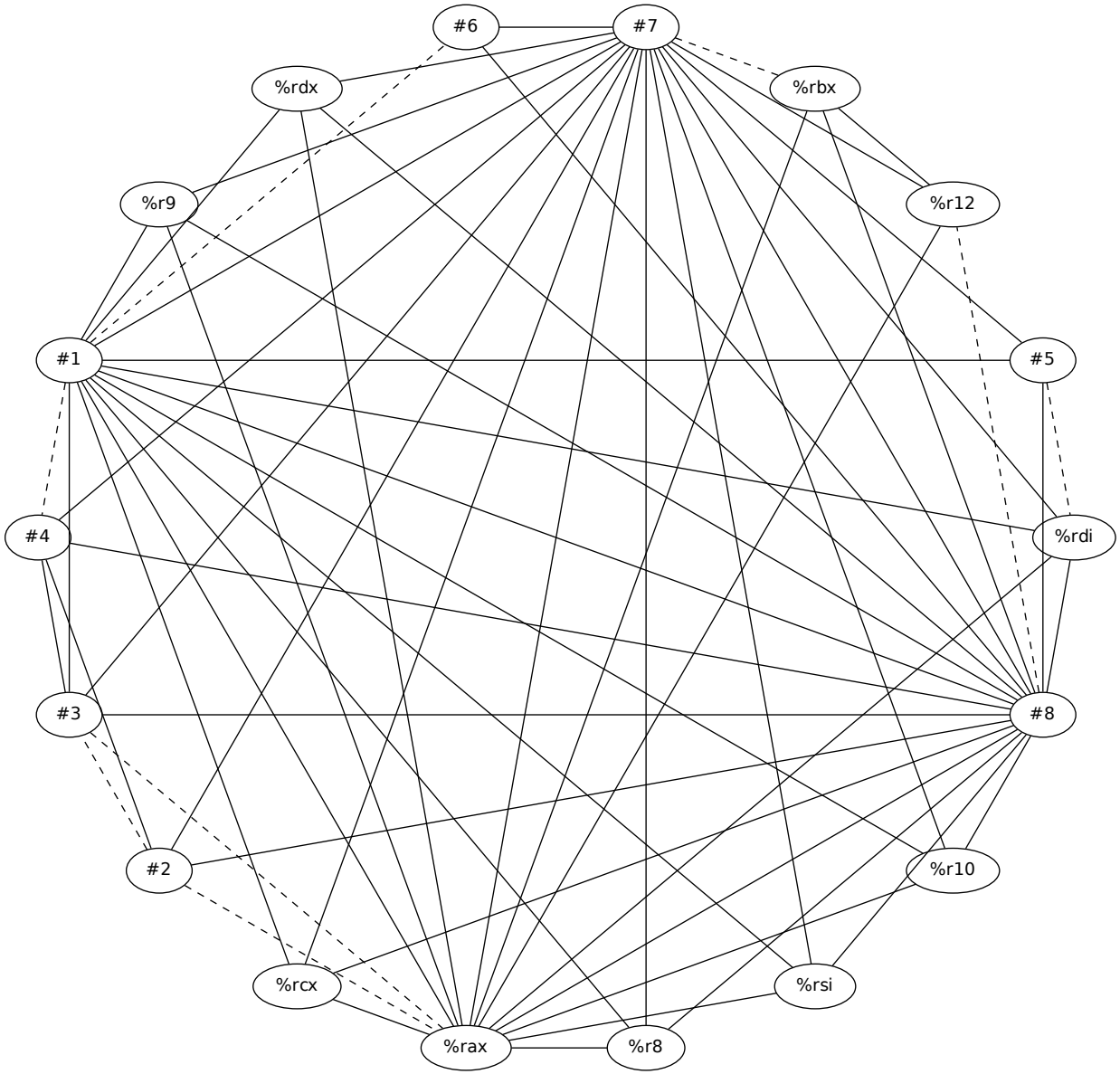
```
val make: Liveness.info Label.map -> graph
```

exemple : factorielle

voici ce que l'on obtient pour la fonction `fact`

10 registres physiques
+
8 pseudo-registres

arêtes de préférence en pointillés



on peut alors voir le problème de l'allocation de registres comme un problème de **coloriage de graphe** :

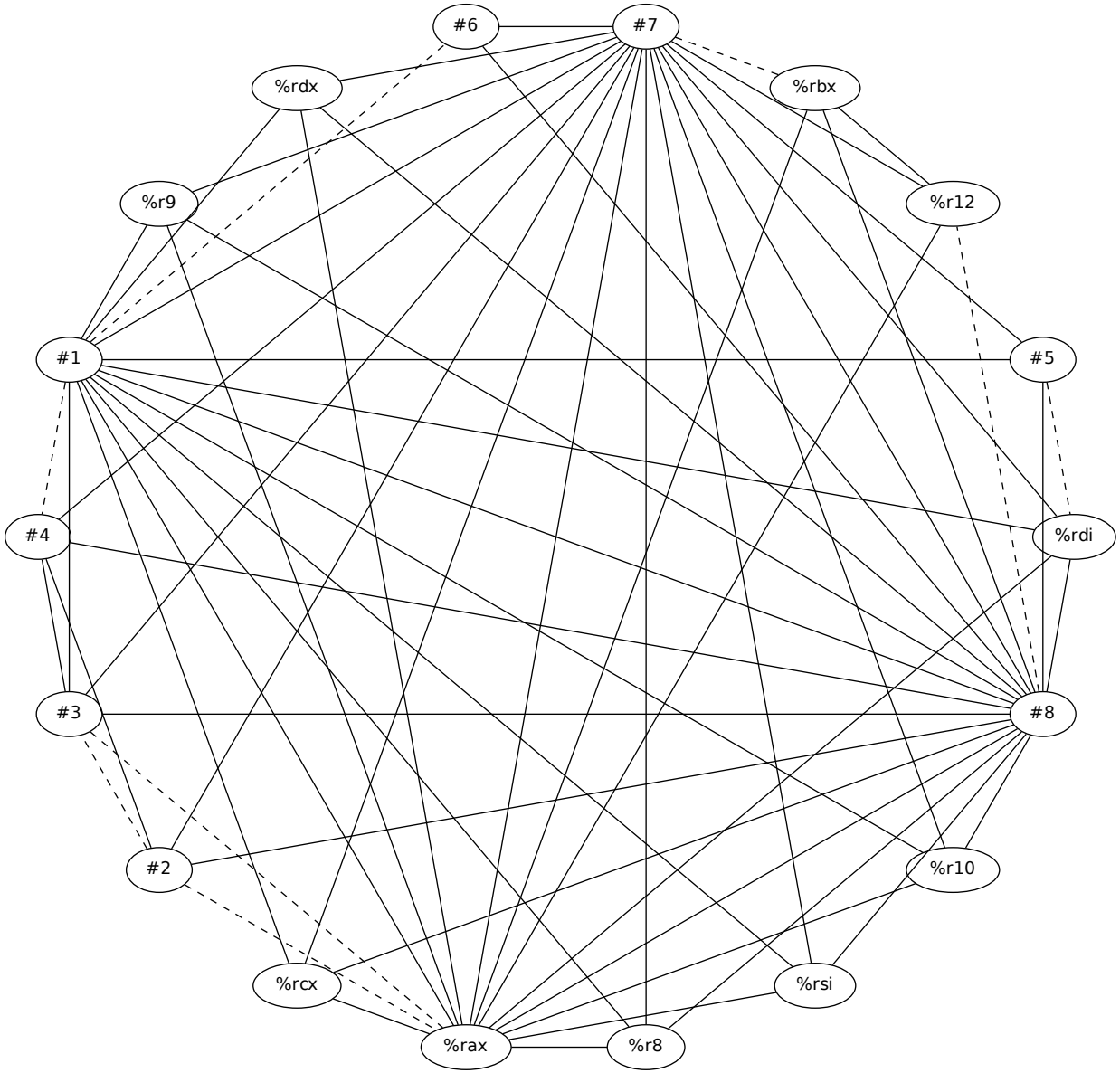
- les couleurs sont les registres physiques
- deux sommets liés par une arête d'interférence ne peuvent recevoir la même couleur
- deux sommets liés par une arête de préférence doivent recevoir la même couleur autant que possible

note : il y a dans le graphe des sommets qui sont des registres physiques, c'est-à-dire des sommets déjà coloriés

exemple de la factorielle

observons les couleurs possibles pour les pseudo-registres

	couleurs possibles
#1	%r12, %rbx
#2	toutes
#3	toutes
#4	toutes
#5	toutes
#6	toutes
#7	%rbx
#8	%r12



sur cet exemple, on voit tout de suite que le coloriage est impossible

- seulement deux couleurs pour colorier #1, #7 et #8
- ils interfèrent tous les trois

si un sommet ne peut être colorié, il correspondra à un emplacement de pile ; on dit qu'on **vide en mémoire** (*spill*) le pseudo-registre

quand bien même le graphe serait effectivement coloriable, le déterminer serait trop coûteux (c'est un problème NP-complet)

on va donc colorier en utilisant des **heuristiques**, avec pour objectifs

- une complexité linéaire ou quasi-linéaire
- une bonne exploitation des arêtes de préférence

l'un des meilleurs algorithmes est dû à George et Appel
(*Iterated Register Coalescing*, 1996)

cet algorithme exploite les idées suivantes

soit K le nombre de couleurs (*i.e.* le nombre de registres physiques)

une première idée, due à Kempe (1879 !), est la suivante : si un sommet a un degré $< K$, alors on peut le retirer du graphe, colorier le reste, et on sera ensuite assuré de pouvoir lui donner une couleur ; cette étape est appelée **simplification**

retirer un sommet diminue le degré d'autres sommets et peut donc produire de nouveaux candidats à la simplification

les sommets retirés sur donc mis sur une pile

lorsqu'il ne reste que des sommets de degré $\geq K$, on en choisit un comme **candidat au vidage** (*potential spill*) ; il est alors retiré du graphe, mis sur la pile et le processus de simplification peut reprendre

on choisit de préférence un sommet qui

- est peu utilisé (les accès à la mémoire coûtent cher)
- a un fort degré (pour favoriser de futures simplifications)

lorsque le graphe est vide, on commence le processus de coloration, appelé **sélection**

on dépile les sommets un à un et pour chacun

- s'il s'agit d'un sommet de faible degré, on est assuré de lui trouver une couleur
- s'il s'agit d'un sommet de fort degré, c'est-à-dire d'un candidat au vidage, alors
  - soit il peut être tout de même colorié car ses voisins utilisent moins de K couleurs ; on parle de **coloriage optimiste**
  - soit il ne peut être colorié et doit être effectivement vidé en mémoire (on parle d'*actual spill*)

enfin, il convient d'utiliser au mieux les arêtes de préférence

pour cela, on utilise une technique appelée **coalescence** (*coalescing*) qui consiste à fusionner deux sommets du graphe

comme cela peut augmenter le degré du sommet résultant, on ajoute un critère suffisant pour ne pas détériorer la K-colorabilité

Définition (critère de George)

Un sommet pseudo-registre v_2 peut être fusionné avec un sommet v_1, si tout voisin de v_1 qui est un registre physique ou de degré $\geq K$ est également voisin de v_2.

De même, un sommet physique v_2 peut être fusionné avec un sommet v_1, si tout voisin de v_1 qui est un pseudo-registre ou de degré $\geq K$ est également voisin de v_2.

le sommet v_1 est supprimé et le graphe mis à jour

pseudo-code de l'algorithme de George-Appel

s'écrit naturellement récursivement

```
let rec simplify g =
  ...
and coalesce g =
  ...
and freeze g =
  ...
and spill g =
  ...
and select g v =
  ...
```

note : la pile des sommets à colorier est donc implicite

```
let rec simplify g =
  if il existe un sommet v sans arête de préférence
     de degré minimal et < K
  then
    select g v
  else
    coalesce g
```

```
and coalesce g =
  if il existe une arête de préférence v1-v2
     satisfaisant le critère de George
  then
    g <- fusionner g v1 v2
    c <- simplify g
    c[v1] <- c[v2]
    renvoyer c
  else
    freeze g
```

```
and freeze g =
  if il existe un sommet v de degré minimal < K
  then
    g <- oublier les arêtes de préférence de v
    simplify g
  else
    spill g
```

```
and spill g =
  if g est vide
  then
    renvoyer le coloriage vide
  else
    choisir un sommet v de coût minimal
    select g v
```

on peut prendre par exemple

$$coût(v) = \frac{nombre\ d'utilisations\ de\ v}{degré\ de\ v}$$

```
and select g v =
  supprimer le sommet v de g
  c <- simplify g
  if il existe une couleur r possible pour v
  then
    c[v] <- r
  else
    c[v] <- spill
  renvoyer c
```

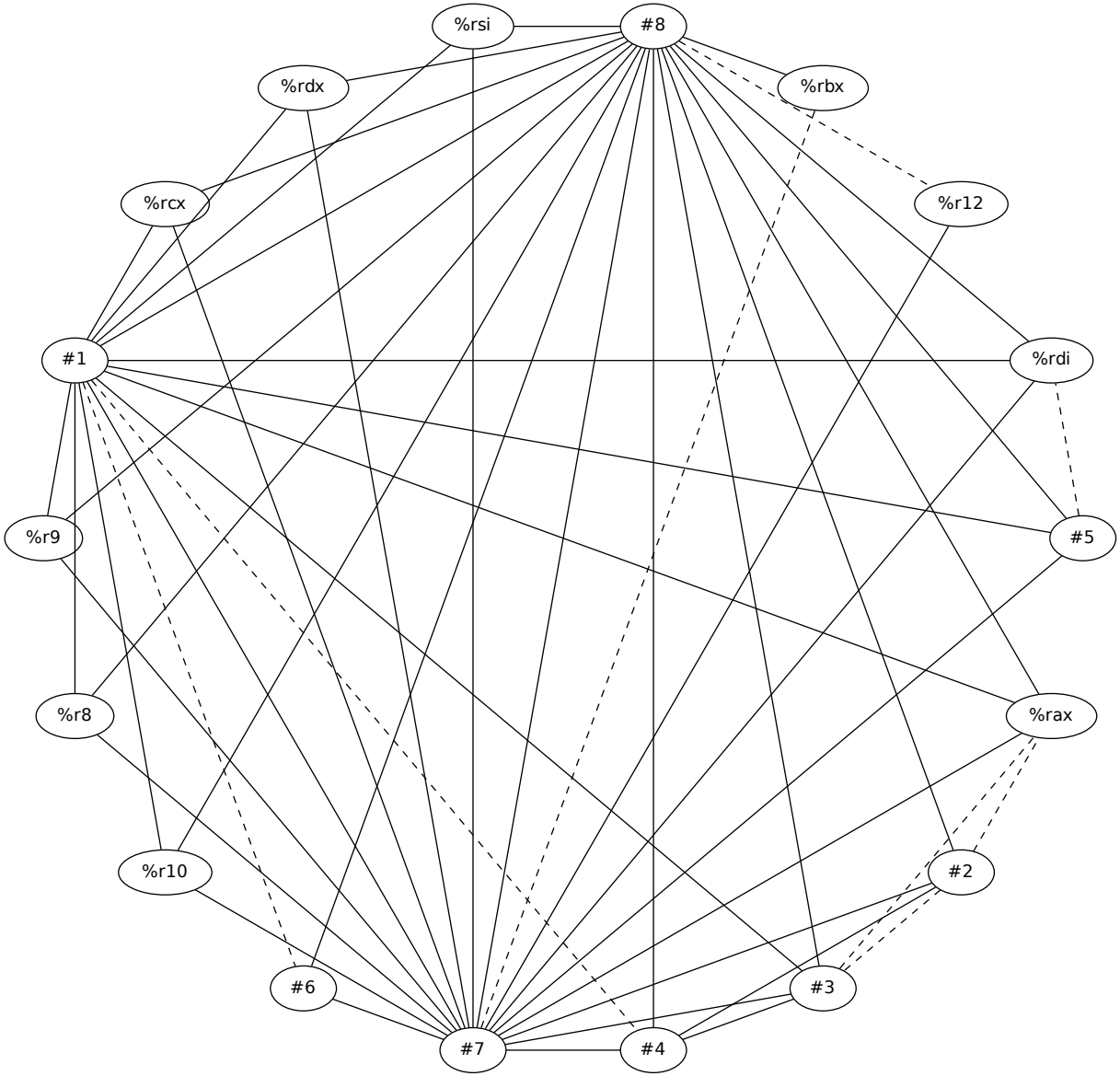
exemple

1.

`simplify g` $\rightarrow$

`coalesce g` $\rightarrow$

sélectionne #2- - -#3



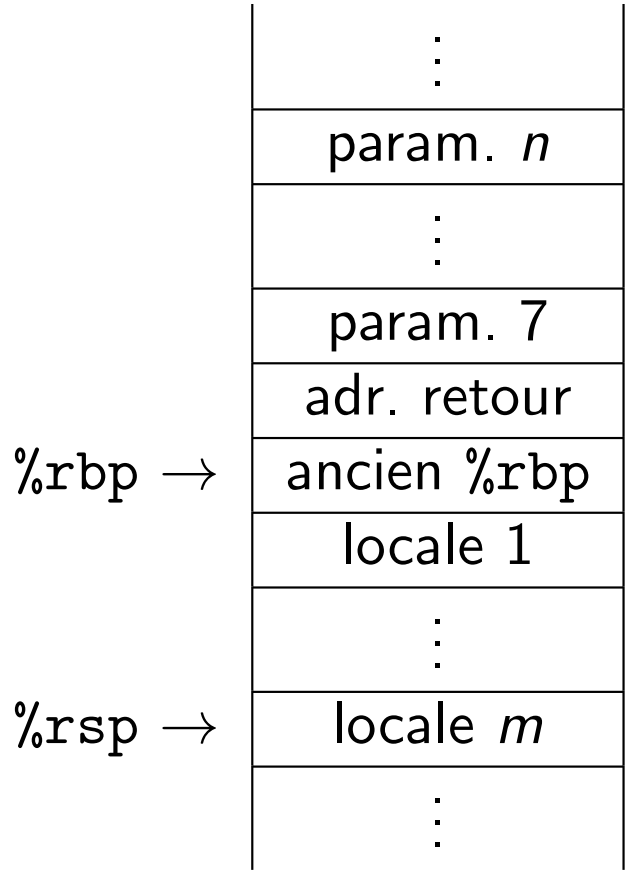
puis on dépile

8.	`coalesce #8- - -%r12`	$\rightarrow$	$c[\#8] = \texttt{\%r12}$
7.	`select #1`	$\rightarrow$	$c[\#1] = \texttt{\%rbx}$
6.	`select #7`	$\rightarrow$	$c[\#7] = \texttt{spill}$
5.	`coalesce #5- - -%rdi`	$\rightarrow$	$c[\#5] = \texttt{\%rdi}$
4.	`coalesce #3- - -%rax`	$\rightarrow$	$c[\#3] = \texttt{\%rax}$
3.	`coalesce #6- - -#1`	$\rightarrow$	$c[\#6] = c[\#1] = \texttt{\%rbx}$
2.	`coalesce #4- - -#1`	$\rightarrow$	$c[\#4] = c[\#1] = \texttt{\%rbx}$
1.	`coalesce #2- - -#3`	$\rightarrow$	$c[\#2] = c[\#3] = \texttt{\%rax}$

et les pseudo-registres vidés ?

que fait-on des pseudo-registres vidés en mémoire ?

on leur associe des emplacements sur la pile, dans la zone basse du tableau d'activation, en dessous des paramètres



plusieurs pseudo-registres peuvent occuper le même emplacement de pile, s'ils n'interfèrent pas ⇒ comment minimiser m ?

c'est de nouveau un problème de coloriage de graphe, mais cette fois avec une infinité de couleurs possibles, chaque couleur correspondant à un emplacement de pile différent

algorithme :

1. on fusionne toutes les arêtes de préférence (coalescence), parce que `mov` entre deux registres vidés coûte cher
2. on applique ensuite l'algorithme de simplification, en choisissant à chaque fois le sommet de degré le plus faible (heuristique)

le résultat de l'allocation de registres a la forme suivante

```
type color = Spilled of int | Reg of Register.t

type coloring = color Register.map
```

la fonction d'allocation se présente alors ainsi

```
val alloc_registers: Interference.graph -> coloring
```

on obtient l'allocation de registres suivante

```
#1 -> %rbx
#2 -> %rax
#3 -> %rax
#4 -> %rbx
#5 -> %rdi
#6 -> %rbx
#7 -> stack -8
#8 -> %r12
```

ce qui *donnerait* le code suivant

```
fact(1)
  entry : L17

  L17: alloc_frame          --> L16
  L16: mov %rbx -8(%rbp)    --> L15
  L15: mov %r12 %r12        --> L14
  L14: mov %rdi %rbx        --> L10
  L10: mov %rbx %rbx        --> L9
  L9 : jle $1 %rbx      --> L8, L7
  L8 : mov $1 %rax          --> L1
  L1 : goto                 --> L22
  L22: mov %rax %rax        --> L21
  L21: mov -8(%rbp) %rbx    --> L20
```

```
L20: mov %r12 %r12   --> L19
L19: delete_frame    --> L18
L18: return
L7 : mov %rbx %rdi   --> L6
L6 : add $-1 %rdi    --> L5
L5 : goto            --> L13
L13: mov %rdi %rdi   --> L12
L12: call fact(1)    --> L11
L11: mov %rax %rax   --> L4
L4 : mov %rbx %rbx   --> L3
L3 : mov %rax %rax   --> L2
L2 : imul %rbx %rax --> L1
```

comme on le constate, de nombreuses instructions de la forme

$$\texttt{mov}\ v\ v$$

peuvent être éliminées ; c'était l'intérêt des arêtes de préférence

ce sera fait pendant la traduction vers LTL

le langage LTL

la plupart des instructions LTL sont les mêmes que dans ERTL, si ce n'est que les registres sont maintenant tous des registres physiques ou des emplacements de pile

```
type instr =
  | Eaccess_global of ident * register * label
  | Eload of register * int * register * label
  | ...
  | Econst of int32 * color * label
  | Emunop of munop * color * label
  | Embinop of mbinop * color * color * label
  | ...
```

par ailleurs, `Ealloc_frame`, `Edelete_frame` et `Eget_param` disparaissent, au profit de manipulation explicite de `%rsp` et `%rbp`

on traduit chaque instruction ERTL à l'aide d'une fonction qui prend en argument le coloriage du graphe d'une part, et la structure du tableau d'activation d'autre part (qui est maintenant connue pour chaque fonction)

```
type frame = {
  f_params: int; (* taille paramètres + adresse retour *)
  f_locals: int; (* taille variables locales *)
}

let instr colors frame = function
  | ...
```

traduction ERTL vers LTL

une variable `r` peut être

- déjà un registre physique
- un pseudo-registre réalisé par un registre physique
- un pseudo-registre réalisé par un emplacement de pile

dans certains cas, la traduction est facile car l'instruction assembleur permet toutes les combinaisons

exemple

```
let instr colors frame = function
  | Ertltree.Econst (n, r, l) ->
      let c =
        try Register.M.find r colors with Not_found -> r in
      Econst (n, c, l)
```

dans d'autres cas, en revanche, c'est plus compliqué car toutes les opérandes ne sont pas autorisées

le cas d'un accès à une variable globale, par exemple,

```
  | Eaccess_global (x, r, l) ->
      ?
```

pose un problème quand `r` est sur la pile car on ne peut pas écrire

```
  movq x, n(%rbp)
```

```
(too many memory references for ‘movq’)
```

il faut donc utiliser un registre intermédiaire
problème : quel registre physique utiliser ?

en adopte ici une solution simple : deux registres particuliers seront utilisés comme registres temporaires pour ces transferts avec la mémoire, et ne seront pas utilisés par ailleurs
(en l'occurrence on choisit ici d'utiliser `%r11` et `%r15`)

en pratique, on n'a pas nécessairement le loisir de gâcher ainsi deux registres ; on doit alors modifier le graphe d'interférence et relancer une allocation de registres pour déterminer un registre libre pour le transfert

heureusement, cela converge très rapidement en pratique
(2 ou 3 étapes seulement)

on se donne donc les deux registres temporaires

```
val tmp1: Register.t
val tmp2: Register.t
```

pour écrire dans la variable `r`, on se donne une fonction `write`, qui prend également en arguments le coloriage et l'étiquette où il faut aller après l'écriture ; elle renvoie le registre physique dans lequel il faut effectivement écrire et l'étiquette où il faut effectivement aller ensuite

```
let write colors r l = match lookup colors r with
  | Reg hr    ->
      hr, l
  | Spilled n ->
      tmp1, generate (Embinop (Mmov, Reg tmp1, Spilled n, l))
```

on peut maintenant traduire facilement de ERTL vers LTL

```
let instr colors frame = function
  | Ertltree.Eaccess_global (x, r, l) ->
      let hwr, l = write colors r l in
      Eaccess_global (x, hwr, l)
  | ...
```

inversement, on se donne une fonction `read1` pour lire le contenu d'une variable

```
let read1 colors r f = match lookup colors r with
  | Reg hr    -> f hr
  | Spilled n -> Embinop (Mmov, Spilled n, Reg tmp1,
                          generate (f tmp1))
```

et on l'utilise ainsi

```
let instr colors frame = function
  | ...
  | Ertltree.Eassign_global (r, x, l) ->
      read1 colors r (fun hwr -> Eassign_global (hwr, x, l))
```

on se donne de même une fonction `read2` pour lire le contenu de deux variables

et on l'utilise ainsi

```
  | Ertltree.Estore (r1, r2, n, l) ->
      read2 colors r1 r2 (fun hw1 hw2 ->
      Estore (hw1, hw2, n, l))
```

on applique un traitement spécial aux instructions `Mmov` et `Mmul`

```
  | Ertltree.Embinop (op, r1, r2, l) ->
    begin match op, lookup colors r1, lookup colors r2 with
    | Mmov, o1, o2 when o1 = o2 ->
        Egoto l
    | _, (Spilled _ as o1), (Spilled _ as o2)
    | Mmul, o1, (Spilled _ as o2) ->
        read1 colors r2 (fun hw2 ->
        Embinop (op, o1, Reg hw2, generate (
        Embinop (Mmov, Reg hw2, o2, l))))
    | _, o1, o2 ->
        Embinop (op, o1, o2, l)
    end
```

on traduit `Eget_param` en terme d'accès par rapport à `%rbp`

```
  | Ertltree.Eget_param (n, r, l) ->
      let hwr, l = write colors r l in
      Embinop (Mmov, Spilled n, Reg hwr, l)
```

on peut enfin traduire `Ealloc_frame` et `Edelete_frame` en terme de manipulation de `%rbp`/`%rsp`

```
| Ertltree.Ealloc_frame l ->
    Epush   (Reg rbp,                                  generate (
    Embinop (Mmov, Reg rsp, Reg rbp,                   generate (
    Emunop  (Maddi (- frame.f_locals), Reg rsp, l)))))
| Ertltree.Edelete_frame l ->
    Embinop (Mmov, Reg rbp, Reg rsp,                   generate (
    Epop    (rbp, l)))
```

lorsque `f_locals` $= 0$, on peut optimiser avec un simple `push`/`pop` du registre `%rbp`

on n'a plus qu'à assembler tous les morceaux

```
let deffun debug f =
  let ln = Liveness.analyze f.fun_body in
  let ig = Interference.make ln in
  let c, nlocals = Coloring.find ig in
  let n_stack_params =
    max 0 (f.fun_formals - List.length Register.parameters) in
  let frame = { f_params = word_size * (1 + n_stack_params);
                f_locals = word_size * nlocals } in
  graph := Label.M.empty;
  Label.M.iter
    (fun l i ->
       let i = instr c frame i in graph := Label.M.add l i !graph)
    f.fun_body;
  { fun_name  = f.fun_name;
    fun_entry = f.fun_entry;
    fun_body  = !graph; }
```

pour la factorielle, on obtient le code LTL suivant

```
fact()
  entry : L17
  L17: push %rbp              --> L24
  L24: mov %rsp %rbp          --> L23
  L23: add $-8 %rsp           --> L16
  L16: mov %rbx -8(%rbp)      --> L15
  L15: goto                   --> L14
  L14: mov %rdi %rbx          --> L10
  L10: goto                   --> L9
  L9 : jle $1 %rbx        --> L8, L7
  L8 : mov $1 %rax            --> L1
  L1 : goto                   --> L22
  L22: goto                   --> L21
  L21: mov -8(%rbp) %rbx      --> L20
  L20: goto                   --> L19
  L19: mov %rbp %rsp          --> L25
  L25: pop %rbp               --> L18
  L18: return
  L7 : mov %rbx %rdi          --> L6
  L6 : add $-1 %rdi           --> L5
  L5 : goto                   --> L13
  L13: goto                   --> L12
  L12: call fact              --> L11
  L11: goto                   --> L4
  L4 : goto                   --> L3
  L3 : goto                   --> L2
  L2 : imul %rbx %rax         --> L1
```

phase 5 : linéarisation

il reste une dernière étape : le code est toujours sous la forme d'un **graphe de flot de contrôle** et l'objectif est de produire du **code assembleur linéaire**

plus précisément : les instructions de branchement de LTL contiennent

- une étiquette en cas de test positif
- une autre étiquette en cas de test négatif

alors que les instructions de branchement de l'assembleur

- contiennent une unique étiquette pour le cas positif
- poursuivent l'exécution sur l'instruction suivante en cas de test négatif

la linéarisation consiste à parcourir le graphe de flot de contrôle et à produire le code x86-64 tout en notant dans une table les étiquettes déjà visitées

lors d'un branchement, on s'efforce autant que possible de produire le code assembleur naturel si la partie du code correspondant à un test négatif n'a pas encore été visitée

dans le pire des cas, on utilise un branchement inconditionnel (`jmp`)

le code x86-64 est produit séquentiellement à l'aide d'une fonction

```
val emit: Label.t -> X86_64.text -> unit
```

on utilise deux tables

une première pour les étiquettes déjà visitées

```
let visited = Hashtbl.create 17
```

et une seconde pour les étiquettes qui devront rester dans le code assembleur (on ne le sait pas au moment même où une instruction assembleur est produite)

```
let labels = Hashtbl.create 17
let need_label l = Hashtbl.add labels l ()
```

la linéarisation est effectuée par deux fonctions mutuellement récursives

- `lin` produit le code à partir d'une étiquette donnée, s'il n'a pas déjà été produit, et une instruction de saut vers cette étiquette sinon

```
val lin: instr Label.map -> Label.t -> unit
```

- `instr` produit le code à partir d'une étiquette et de l'instruction correspondante, sans condition

```
val instr: instr Label.map -> Label.t -> instr -> unit
```

la fonction `lin` est un simple parcours de graphe

si l'instruction n'a pas déjà été visitée, on la marque comme visitée et on appelle `instr`

```
let rec lin g l =
  if not (Hashtbl.mem visited l) then begin
    Hashtbl.add visited l ();
    instr g l (Label.M.find l g)
```

sinon on marque son étiquette comme requise dans le code assembleur et on produit un saut inconditionnel vers cette étiquette

```
  end else begin
    need_label l;
    emit (Label.fresh ()) (jmp l)
  end
```

la fonction `instr` produit effectivement le code x86-64 et rappelle récursivement `lin` sur l'étiquette suivante

```
and instr g l = function
  | Econst (n, r, l1) ->
      emit l (movq (imm32 n) (operand r)); lin g l1
  | Eaccess_global (x, r, l1) ->
      emit l (movq (lab x) (register r)); lin g l1
  | ...
```

le cas intéressant est celui d'un branchement (on considère ici `Emubranch` ; c'est identique pour `Embbranch`)

on considère d'abord le cas favorable où le code correspondant à un test négatif n'a pas encore été produit

```
  | Emubranch (br, r, lt, lf)
    when not (Hashtbl.mem visited lf) ->
      need_label lt;
      emit l (ubranch br r lt);
      lin g lf;
      lin g lt
```

(où `ubranch` est la fonction qui produit les instructions x86-64 de branchement, à savoir `testq`/`cmpqq` et `j`*cc*)

sinon, il est possible que le code correspondant à un test positif n'ait pas encore été produit et on peut alors avantageusement **inverser la condition** de branchement

```
  | Emubranch (br, r, lt, lf)
    when not (Hashtbl.mem visited lt) ->
      instr g l (Emubranch (inv_ubranch br, r, lf, lt))
```

où

```
let inv_ubranch = function
  | Mjz  -> Mjnz
  | Mjnz -> Mjz
  | ...
```

enfin, dans le cas où le code correspondant aux deux branches a déjà été produit, on n'a pas d'autre choix que de produire un branchement inconditionnel

```
  | Emubranch (br, r, lt, lf) ->
      need_label lt; need_label lf;
      emit l (ubranch br r lt);
      emit l (jmp lf)
```

note : on peut essayer d'estimer la condition qui sera vraie le plus souvent

le code contient de nombreux `goto` (boucles `while` dans la phase RTL, insertion de code dans la phase ERTL, suppression d'instructions `mov` dans la phase LTL)

on élimine ici les `goto` lorsque c'est possible

```
  | Egoto l1 ->
      if Hashtbl.mem visited l1 then begin
        need_label l1;
        emit l (jmp l1)
      end else begin
        emit l nop; (* sera en fait éliminé *)
        lin g l1
      end
```

le programme principal enchaîne toutes les phases de la compilation

```
let f = open_in file in
let buf = Lexing.from_channel f in
let p = Parser.file Lexer.token buf in
close_in f;
let p = Typing.program p in
let p = Is.program p in
let p = Rtl.program p in
let p = Ertl.program p in
let p = Ltl.program p in
let code = Lin.program p in
let c = open_out (Filename.chop_suffix file ".c" ^ ".s") in
let fmt = formatter_of_out_channel c in
X86_64.print_program fmt code;
close_out c
```

et voilà !

```
fact:   pushq %rbp
        movq  %rsp, %rbp
        addq  $-8, %rsp
        movq  %rbx, -8(%rbp)
        movq  %rdi, %rbx
        cmpq  $1, %rbx
        jle   L8
        movq  %rbx, %rdi      ## inutile, dommage
        addq  $-1, %rdi
        call  fact
        imulq %rbx, %rax
L1:
        movq  -8(%rbp), %rbx
        movq %rbp, %rsp
        popq %rbp
        ret
L8:
        movq  $1, %rax
        jmp   L1
```

on pouvait faire un peu mieux à la main

```
fact:   cmpq    $1, %rdi        # x <= 1 ?
        jle     L3
        pushq   %rdi            # sauve x sur la pile
        decq    %rdi
        call    fact            # fact(x-1)
        popq    %rcx
        imulq   %rcx, %rax      # x * fact(x-1)
        ret
L3:
        movq    $1, %rax
        ret
```

mais il est toujours plus facile d'optimiser **un** programme

taille de mini-C

	lignes de code
parsing	227
typage	170
sélection d'instruction	103
RTL	203
ERTL	185
LTL	189
durée de vie	106
interférence	60
coloriage	220
linéarisation	151
assembleur	253
divers	124
total	**1991**

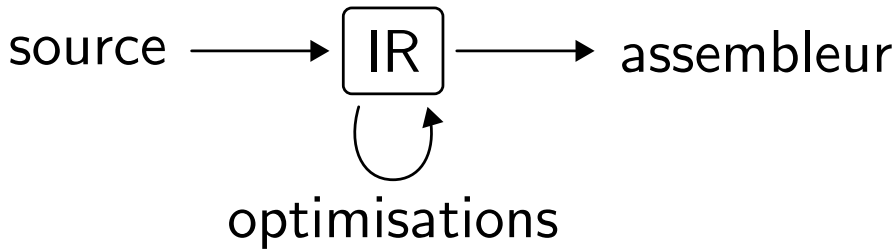
- l'architecture présentée ici est celle de **CompCert**
  - les optimisations sont réalisées au niveau RTL
- le compilateur **gcc** intercale un langage SSA (cf plus loin)

$$\text{partie avant} \rightarrow \text{SSA} \rightarrow \text{RTL} \rightarrow \cdots$$

  et les optimisations sont réalisées au niveau SSA et au niveau RTL
- le compilateur **clang** se repose sur LLVM

il s'agit d'une infrastructure pour aider à la construction de compilateurs optimisant

LLVM propose un langage intermédiaire, IR, et des outils d'optimisation et de compilation de ce langage



le compilateur C `clang` est construit sur LLVM

on peut obtenir le code IR avec

```
> clang -O1 -c -emit-llvm fact.c -o fact.bc
```

et le rendre lisible avec

```
> llvm-dis fact.bc -o fact.ll
```

code IR de `fact`

```
define i32 @fact(i32) {
  %2 = icmp slt i32 %0, 2
  br i1 %2, label %10, label %3
; <label>:3:                                     ; preds = %1
  br label %4
; <label>:4:                                     ; preds = %3, %4
  %5 = phi i32 [ %7, %4 ], [ %0, %3 ]
  %6 = phi i32 [ %8, %4 ], [ 1, %3 ]
  %7 = add nsw i32 %5, -1
  %8 = mul nsw i32 %5, %6
  %9 = icmp slt i32 %5, 3
  br i1 %9, label %10, label %4
; <label>:10:                                    ; preds = %4, %1
  %11 = phi i32 [ 1, %1 ], [ %8, %4 ]
  ret i32 %11
}
```

le langage IR ressemble beaucoup à notre langage RTL

- des pseudo-registres (%2, %5, %6, etc.)
- un graphe de flot de contrôle
- des appels encore haut niveau

mais il y a aussi des différences notables

- c'est un langage typé
- le code est **en forme SSA** (*Single Static Assignement*) : chaque variable n'est affectée qu'une fois

bien entendu, le code d'origine est susceptible d'affecter plusieurs fois une même variable

on recourt alors à un opérateur appelé Φ pour réconcilier plusieurs branches du flot de contrôle

ainsi,

```
%5 = phi i32 [ %7, %4 ], [ %0, %3 ]
```

signifie que %5 reçoit la valeur de %7 si on vient du bloc %4 et la valeur de %0 si on vient du bloc %3

l'intérêt de la forme SSA est que l'on peut maintenant

- **attacher** une propriété à chaque variable
  (par ex. valoir 42, être positif, être dans l'intervalle [34,55], etc.)
- l'exploiter ensuite **partout** où cette variable est utilisée

la forme SSA facilite bon nombre d'optimisations

on obtient de l'assembleur avec le compilateur LLVM

```
> llc fact.bc -o fact.s
```

cette phase inclut

- l'explicitation des conventions d'appel ($\approx$ ERTL)
- l'allocation de registres ($\approx$ LTL)
- la linéarisation

en particulier, c'est l'allocation de registres qui va faire disparaître la majeure partie des opérations Φ (mais quelques `mov` peuvent néanmoins être nécessaires)

```
> llc fact.bc -o fact.s
```

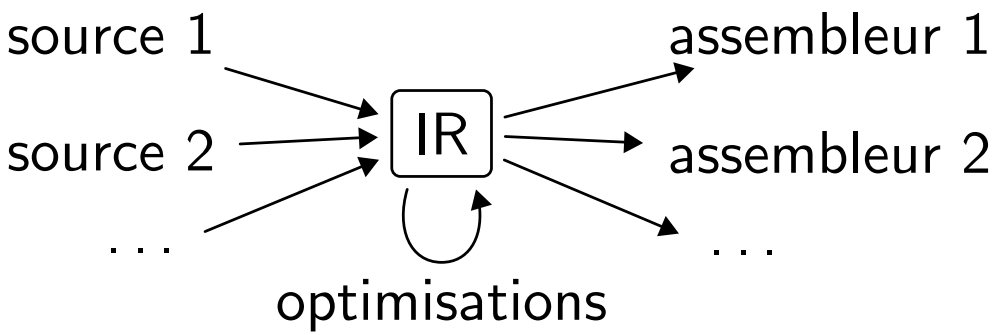
```
fact:   movl  $1, %eax
        cmpl  $2, %edi
        jl    L3
L2:     imull %edi, %eax
        leal  -1(%rdi), %ecx
        cmpl  $2, %edi
        movl  %ecx, %edi
        jg    2
L3:     ret
```

on peut avantageusement tirer partie de LLVM pour

- écrire un nouveau compilateur pour un langage S en se contentant d'écrire la partie avant et la traduction vers IR

et/ou

- concevoir et réaliser de nouvelles optimisations, sur le langage IR



la suite

- TD d'aide au projet aujourd'hui et jeudi 7 janvier
- dernier cours le 8 janvier
- projet à rendre pour le dimanche 17 janvier avant 18:00